The Report Committee for Joseph Sory Forbes
certifies that this is the approved version of the following report:

A Case Study of Continuous Delivery of a Mobile Application

APPROVED BY

SUPERVISING COMMITTEE:

Adnan Aziz, Supervisor

Sarfraz Khurshid

A Case Study of Continuous Delivery of a Mobile Application

by

Joseph Sory Forbes, B.A.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2015

A Case Study of Continuous Delivery of a Mobile Application

Joseph Sory Forbes, M.S.E.
The University of Texas at Austin, 2015

Supervisor: Adnan Aziz

The concept of continuous delivery extends the existing concept of continuous integration beyond its traditional scope of development, build, and test into deployment to end-users. This project is an implementation of continuous delivery for an Android application. In addition to deployment to client devices through the Google Play Store, the implemented continuous delivery "pipeline" uses integration of Google Analytics into the Android application and the continuous delivery pipeline itself to automate staging from alpha release to beta release based on observed manual tester activity.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Continuous delivery is an extension of the continuous integration approach to automating build and test activities within a software development organization. Continuous delivery adds steps downstream from build and test to carry software changes all the way from the developer's desk to the end user with as little human intervention as possible [5]. The authors of *Continuous Delivery* visualize continuous delivery as a pipeline made of up discrete stages, each of which validates the outputs of prior stages and prevents unreleasable code from reaching subsequent stages.

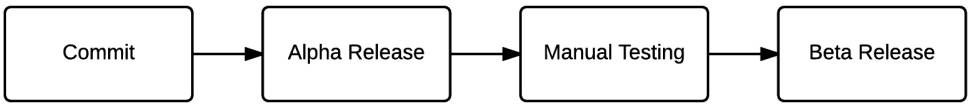

Figure 1.1: A Continuous Delivery Pipeline for an Android Application

Two factors distinguish continuous delivery of Android applications from continuous delivery of other types of software products: the Google Play Store and the standard Android build system. The Google Play Store, which is the standard mechanism for deployment of consumer applications to Android

devices, introduces requirements and constraints that must be addressed in order for deployment tasks to be automated. The standard Android build system has not yet stabilized since its migration from Maven to Gradle, which started in 2013. This continuing evolution presents challenges to automation of the build and test tasks within the continuous delivery system.

In this paper, I will describe work I have done to implement a continuous delivery pipeline for an Android application. This continuous delivery pipeline makes use of instrumentation in the Android application to measure manual tester activity in the application and automatically promote the application from "alpha" to "beta" when sufficient manual testing activity has occurred.

1.1 Motivation

The interesting parts of most consumer-targeted software are the capabilities and attributes the software presents to the user. Time spent performing tasks other than developing the software is time not spent adding to or improving the user-visible functionality. The goal of continuous delivery is to reduce the time and effort needed deliver reliable, working, cost-effective software to end users [5]. This includes the performance of all necessary activities.

The inclusion of release and deployment activities in a continuous delivery process ensures that the team identifies and addresses deployment considerations early in the development process. The automation of release activities reduces the effort to deploy new versions of the product. This in turn allows

the organization to deploy more often, getting feedback to the business as well as to the developers more rapidly. Early and frequent release of small units of functionality has the effect of increasing the organization's agility [6] and deepening its collective understanding of the use cases and performance considerations that most affect the product's success.

1.2 Vision

The goal of this project is to develop a continuous delivery pipeline for an Android application that substantially automates the process of delivering tested Android application software to a group of beta testers.

1.3 User Perspectives

The context of a continuous delivery pipeline is a flow of software and information about software among people playing various roles in the delivery process. Each of these roles has a distinct set of responsibilities within the system. Each role also has a distinct set of expectations of the system and the other participants in it. These responsibilities and expectations, described in the sections below, form the foundation of the continuous delivery system's requirements.

The participants in the delivery process typically do not want to spend a lot of time building, packaging, testing, and deploying software, ascertaining delivery status, or troubleshooting problems in any of these areas. Time saved

by automating these tedious activities is available for making the contributions to the delivery process that each participant is responsible for. Wherever possible, the participants would like to push tedious tasks off to the continuous delivery system.

The roles used in this section are drawn from the Scrum product management methodology, with the addition of a "Development Operations" role that is starting to be found in organizations that use continuous integration and continuous delivery approaches. Of the roles described below, two or three might reasonably be expected to be played by the same people, due to the self-organizing nature of Scrum teams. The responsibilities and expectations described below attach to the role and not the person; when a team member is performing manual testing activities, he or she has the responsibilities and expectations of a manual tester.

1.3.1 Product Owner

In the Scrum software development methodology, the Product Owner is a single person [11, p. 34] with responsibility and accountability for representing the interests of the customers and users [13, p.177, 178]. Among the Product Owner's responsibilities are:

- Prioritizing the work to be done by the development team.
- Judging whether implemented functionality is acceptable.
- Determining and communicating release dates and contents.

In the context of continuous delivery, the Product Owner's primary expectation of the system is that the Product Owner's decisions about what to release and when will be executed without undue delay or difficulty [10, p.12]. Additional desires of the Product Owner are the ability to get status information about release activities and access data relating to usage and performance of the system.

1.3.2 Developer

The Developer role, first and foremost, is responsible for creating and modifying the source code and other information assets that are used to build the deliverable product. With Scrum and other modern software development methodologies' emphasis on test automation and short release cycles, developers on many modern development teams are also expected to write and execute automated tests of the modules being created or maintained. The tests written by developers are executed by the developer writing them and by later developers maintaining the covered modules and adjacent modules. These tests are often also executed as part of the immediate post-build verification process.

When relying on a continuous delivery system to perform build, test, and packaging tasks, the developer needs to be informed as quickly as possible if the build has been broken by a commit. Reasons for failures that are of interest to the developer include: product cannot be built in the controlled build environment, automated tests that were not run prior to the commit fail, and failures of source or executable code and tests to meet other requirements

such as code coverage or code style validation.

1.3.3 Manual Tester

Even on a project with excellent automated tests, it is generally advisable to have human testers manually test the Application, however briefly, to verify that there is not some catastrophic issue with the software that is not detectable by the automated tests. It is the Manual Tester's responsibility to execute the manual tests properly and see that the results are made available for analysis.

The Manual Tester's needs from the system are relatively simple. The person playing this role would like to be able to get new versions quickly and easily. The Manual Tester would also like to spend as little time as possible tracking the tests executed and their results.

1.3.4 Development Operations

The role of Development Operations is new relative to the Developer and Manual Tester, and even to the Product Owner role. The Development Operations role has grown out of the complexity of the automation and configuration of continuous integration and continuous delivery systems and the infrastructure to support them. It is the responsibility of Development Operations to install, maintain, and administer the software systems that make up the continuous delivery pipeline. This might include set up and administration of accounts and data on various hosted services, such as an online source code

repository.

As the primary maintainer of the continuous delivery pipeline, the Development Operations role has many expectations of the various components that make up the pipeline. For the purpose of this project, we will consider only the Development Operations role's expectations regarding system status. The person playing the Development Operations role wants to be notified quickly of the failure of any job running on the continuous integration server, so that any corrective measures or measures to prevent cascading problems can be taken. The Development Operations role also would like to be able to see the status of recent jobs on the continuous integration server, and be able to quickly distinguish between correct operation and incorrect operation.

1.4 Structure of Report

Chapter 2 provides background on continuous delivery, the technologies used by the implementation that is the subject of this report, and the subject Android application used in this project. Chapter 3 provides an overview of the implementation of the pipeline. Chapter 4 discusses results and lessons learned. Chapter 5 provides some conclusions and ideas for further work in this area.

Chapter 2

Background

The primary technologies I used to build my continuous delivery pipeline are the Jenkins continuous integration server, Google Analytics, and Google Play Services. This chapter provides summary background for continuous delivery and for each of these primary technologies, and provides a brief overview of the application used to exercise the continuous delivery pipeline.

2.1 Continuous Delivery

As mentioned in the introduction, a continuous delivery system is often visualized as a series of stages arranged into a continuous delivery pipeline. The figure and descriptions below describe the stages of a generic continuous delivery pipeline based on the ideas in [5].

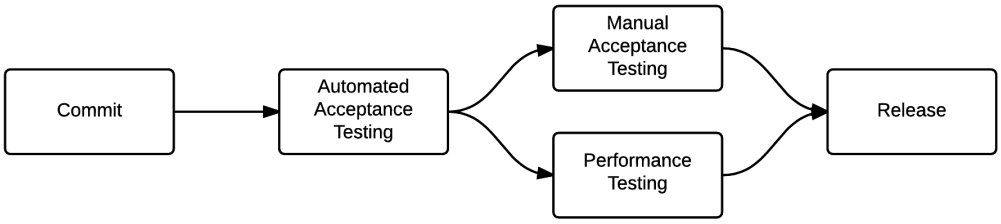

Figure 2.1: A Generic Continuous Delivery Pipeline

2.1.1 Commit Stage

The commit stage describes things that happen on every commit to a shared source code repository. During this stage, the product is built and unit tests are run. Static analysis of the code and analysis of test outputs are performed. Based on the principle that we should test what we deploy and deploy what we test, the binaries built in the commit stage should be used for all subsequent stages, through to production deployment [5].

The primary purpose of the commit stage is to ensure buildability of the production-bound binaries and perform a quick verification of the newly-built version's quality. While the commit stage testing and verification does not need to be as fast as tests run by a developer during development, they should be quick enough to provide timely feedback to development that a given commit to the source code repository has broken the build.

2.1.2 Automated Acceptance Testing Stage

The automated acceptance test stage acts as a cost-effective gatekeeper, preventing faulty products from flowing downstream to the more resource-intensive manual testing and performance testing stages. The tests run in the commit stage are generally largely based on the unit and partial integration tests written by developers for use prior to committing changes. These tests are designed to verify that the various components of the product work as their implementers expect them to. The tests run during the automated acceptance stage are designed to verify that the product as a whole works as its end-users

expect it to [5].

It should be noted that, somewhat in conflict to the single-threaded "pipeline" paradigm, testing stages downstream from the automated acceptance test stage may be run in parallel. This is due to the separate emphases and relatively long times taken to perform these later-stage tests. By allowing parallel performance of these tests, the continuous delivery pipeline is able to get products through many types of time-consuming tests in the smallest amount of "wall" or "calendar" time.

2.1.3 Manual Testing Stage

In the manual testing stage, human testers interact with the application. For a product with highly developed automated acceptance tests, this might be a simple smoke test to ensure that nothing catastrophic has gone wrong to prevent the product from being properly deployed. For products with less-developed automated acceptance tests, the manual testing stage might involve carefully following a complex script of actions to be taken and application behaviors to be verified. The manual tester's interests should drive the design of this stage, emphasizing test result management and ease of deployment.

2.1.4 Performance Testing Stage

Performance tests ensure that the system performs as expected under a range of conditions. The authors of *Continuous Delivery* concentrate on capacity testing, that is, testing to ensure that the product is able to handle

the volume and complexity of data, processing, and communications that it is expected to encounter in production [5]. Other types of testing that might occupy the same position in a continuous delivery pipeline include reliability tests, tests of disaster recovery scenarios, and device compatibility tests.

2.1.5 Release Stage

In the release stage, the software is deployed or otherwise made available to customers and end-users. Release stage considerations include data upgrade and conversion, rollback, and staggered rollout. It is the goal of the continuous delivery system to render the production release, formerly an event of singular fear, uncertainty, and doubt within many software organizations, into a non-event. [5].

2.2 Jenkins

In 2010, Jenkins was created as a fork of the Hudson continuous integration server project. Hudson development has continued with support from Oracle. Continuous integration products like Hudson and Jenkins were created to automate and centralize the build, deployment, test, and packaging tasks that development teams are responsible for. A fuller description of Jenkins' history and use as well as best practices for management of Jenkins servers and job definitions can be found in [12]. The core capabilities of a continuous integration server are:

- Source control integration

- Build automation
- Automated testing
- Publishing

In addition to and support of these functions the continuous integration system must provide appropriate and timely information to the stakeholders. To address this need, Jenkins supports both the "pushing" of notifications (for instance, emails) containing status information and "pulling" of status information through the use of dashboards.

As a popular open-source product, Jenkins has gathered a large body of independently-produced plugins to perform various specialized functions. Plugins are used to perform nearly every function, from source control, to build, to notifications and reporting.

Jenkins organizes work to be done into Jobs. Each job can be associated with a separate location in a working directory called a "workspace". Jobs can also share a workspace. Each job has three parts: configuration, build, and post-build. Jobs can be set to run on a schedule, they can be started from other jobs, and they can be started by calls to the Jenkins server's web API made by source control systems or other external applications.

Job configuration covers source control configuration, build environment setup, and a list of events that will trigger the job to build. When a build is triggered, the configuration is read and the environment set up according to the configuration. If the job specifies source control, the workspace

is updated with the latest revision of the source code. If any of these build preparation tasks fails, the build ends, reporting failure.

After the build preparation tasks have been performed, the Jenkins job performs the build tasks one at a time, in the order specified. If any task fails, the build is immediately ended reporting failure. There are several types of build tasks. Common tasks include invocation of a build system such as `make` or Gradle and execution of shell scripts. Through the use of plug-ins, tasks can be made to run only under specific conditions. Shell scripts can report failure by exiting with a non-zero return value.

Lastly, whether the build was successful or not, a Jenkins job has a series of post-build actions. These include such things as publishing to online package repositories, uploading to source control services, and notifying interested parties of the outcome of the build. As with the build tasks, the post-build tasks are run one at a time, in order. An important difference between the build and post-build phases of a Jenkins job is that, in the post-build phase, failure of a given task may or may not stop subsequent post-build tasks from running or result in the build being marked as failed. This behavior depends on the type of the post-build task that failed and may not always be easy to determine without experimentation.

2.3 Google Play Services

The standard app store for Android is the Google Play Store. The Google Play Store allows users to find, install, and upgrade Android applica-

tions. Developers wishing to publish applications to the Google Play Store can do so by creating a complete and valid listing in the Play Store and uploading an Android application package ("APK") file. The Google Play Store has release management features that support three separate "tracks" to allow limited release for testing and evaluation: alpha, beta, and production. Production deployment can be applied in a staged manner in which increasing percentages of users are offered the upgrade over time.

Google supports programmatic interactions with the Google Play Services through the Google Play Developer API. Samples for use of the Google Play Developer API are available for Java and Python. In addition to APK upload, the Google Play Developer API supports other transactions allowing the developer to query for information about the listing and associated release tracks and APK files.

2.4 Google Analytics

Google Analytics is a tool for aggregating and analyzing web application user activity data. Google Analytics is based on software built by Urchin, which Google acquired in 2005 [2]. Primarily used to optimize website design and improve advertising conversion rates, Google Analytics works by allowing the web developer to embed calls to Google Analytics APIs, recording "hits" on various pages and interface components. The data is made available on a website, and through an API. Google supports the integration of Google Analytics and Android applications, enabling the analysis of Android application

usage patterns.

Event data is aggregated and distributed across the Google platform. Distribution of event data can take several hours in some cases, although delays on the order of a few minutes to half an hour are more typical. For each "property" or application, events can be reported and visualized based on the data sent with each event, including application names and versions, screen or page name, and manually-set event properties. Google also makes this data available through an API, offering client libraries to assist development in Java and Python, as well as many other languages. Thorough description of the Google Analytics Service and its use, primarily concentrated on its use for website analytics, but including some information on use of the Core Reporting API in offline applications, can be found in [3].

2.5 LanIr Android Product Background

Continuous delivery assumes that there is a product to be delivered. The product I used for this project is the Android client I had written for a previous project. This previous project is called "LanIr". The name LanIr combines LAN, representing Local Area Network, and IR, representing infrared. The LanIr project allows an Android device user to control media devices such as televisions and DVD players that work with infrared remote controls. The LanIr Android application connects to a LanIr appliance using a web interface running on Node.js. The LanIr Android application can request a list of controllable devices and lists of available commands for each device.

LanIr Android dynamically adds appropriate buttons to Android Activities to allow the user to select a device to control and to issue commands. The LanIr Android application responds to user button presses by sending commands to the LanIr appliance, which the LanIr appliance sends as an infrared remote control commands using the Linux Infrared Remote Control daemon (lircd).

The LanIr Android application has few dependencies on other systems. The LanIr appliance implements a stable and simple API, and the current version of the LanIr Android application does not store user data on any external device. Users interact directly with LanIr, making tracking user activity relatively simple. These factors simplify the work of developing, testing, and troubleshooting the continuous delivery pipeline.

Chapter 3

Implementation

A continuous delivery system can be visualized as a pipeline made up of discrete stages, each of which validates the outputs of prior stages and prevents unreleasable code from reaching subsequent stages. In addition to the ideas presented in *Continuous Delivery*, one can refer to *Hand-on Mobile App Testing* for ideas as to what to include in continuous integration and delivery of a mobile application [8]. For the purposes of this project, I have implemented a continuous delivery pipeline that has the following stages:

1. Commit
2. Alpha Release
3. Manual Testing
4. Beta Release

The context of this pipeline is envisioned as a small software organization using Scrum or a similar project management approach. The Product Owner could decide to start the process of releasing the product at the end of any iteration. Prior to this decision on the part of the Product Owner, testing

is performed either without use of a device or by manually deploying the product to the device. Once the Product Owner has requested release of a given version, the commit stage for that version is ended. The product is released through the Play Store to a group of alpha testers. In the envisioned scenarios, these testers would be a mix of development, product management, and marketing personnel from inside the organization, possibly to include friendly customers or other outsiders. Once a sufficient amount of successful alpha testing activity had taken place, the product would be released to a larger group of beta users, most of whom would be outsiders to the organization.

Subsequent stages of the software delivery life cycle are outside the scope of this project. The current assumption is that promotion from beta to general availability or "production" will be made on the basis of roadmap considerations, analysis of Google Analytics data generated while the version is in beta, and other factors. The infrastructure built to perform automatic promotion to beta release can also be used to perform the general availability release, as a manually triggered activity.

This continuous delivery pipeline does not contain automated acceptance testing, performance testing, or final release to production. Discussion of possible implementations of these stages is included in Chapter 5 at the end of the report.

3.1 Overview of Stages

Each stage's implementation involved one or more Jenkins jobs, except the Manual Testing stage, the implementation of which was in the Android application. The Alpha Release, Manual Testing, and Beta Release stages each involve some degree of integration with Google Play Services and Google Analytics. Custom integration code was required to the implement Google service integration in the Manual Testing and Beta Release stages. The following sections describe each stage at a high level and indicate the Jenkins jobs involved in implementing each stage, as well related notifications. Tables 3.1, 3.2, and 3.3 summarize information about the mappings from continuous delivery stages to Jenkins jobs, the Google Services integrations needed by each job, and the notifications sent by each job on success or failure.

3.1.1 Commit Stage

In the commit stage, the continuous delivery pipeline retrieves, builds, and tests the application. It measures the code coverage, and rehearses preparation of the APK for release. Results of the build are not archived or published. During the commit stage, the product is marked as in development by the inclusion of a "dev" suffix on its "versionName" property. Failures at this stage are reported to the developers and to development operations. The commit stage is implemented in a single Jenkins job, which we will call the "Main" job. The main job is triggered by a notification from the source code repository that is sent whenever a change is committed.

3.1.2 Alpha Release Stage

The alpha stage starts when the product owner requests release of a given version. The product's version information is changed to remove the "dev" suffix. The new version of the product is rebuilt, tested, packaged, and released to the Google Play store, assigned to the "alpha" track. Finally, the product's version information is changed, incrementing the product's version numbers and restoring the "dev" suffix to its versionName value. Failures during the alpha release stage should be reported to the product owner and to development operations. Successful release of a version to the alpha track should be reported to the product owner and to the manual testers. This stage is implemented by four Jenkins jobs, as shown in 3.1.

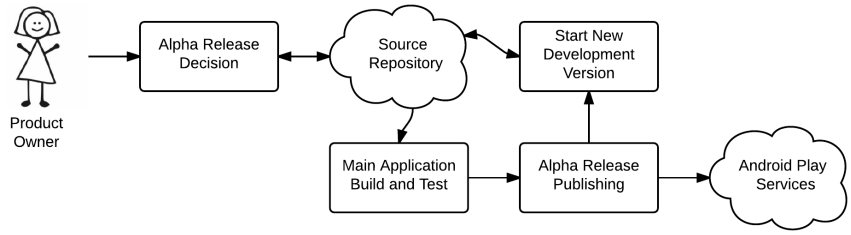

Figure 3.1: Alpha Release Stage Overview

3.1.3 Manual Testing Stage

During the manual testing stage, the new version of the product is deployed to the device. The manual tester interacts with the new version, and the application reports this usage to the Google Analytics service. This stage is implemented by adding Google Analytics API calls to the application. These

interactions are shown on the left-hand side of Figure 3.2. No notifications are sent from the manual testing stage; notifying the other participants of issues in this stage is the responsibility of the manual tester.

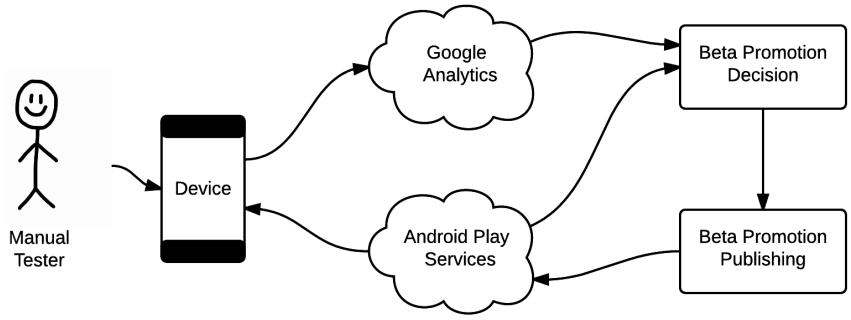

Figure 3.2: Overview of Manual Testing and Beta Release Stages

3.1.4 Beta Release Stage

The beta release stage, shown on the right-hand side of Figure 3.2, starts when the Google Analytics service returns usage information meeting the criteria for beta release. This is detected by periodically calling the necessary Google Analytics and Google Play Developer APIs. When a version meeting the beta release criteria is identified, a second Jenkins job promotes it to the beta track. Failures during the beta release stage should be reported to development operations. Successful promotion of a version to the beta track should be reported to the product owner.

Stage	Jenkins Job(s)
Commit	Main
Alpha Release	Alpha Release Decision, Main, Alpha Release Publishing, Start New Development Version
Manual Testing	(Implemented in Android application.)
Beta Release	Beta Promotion Decision, Beta Promotion

Table 3.1: Summary of Stages and Jobs

Job	Success Notification	Failure Notification
Main	–	Developer, Product Owner,* Development Operations
Alpha Release Decision	–	Product Owner, Development Operations
Alpha Publishing	Product Owner, Manual Tester	Product Owner, Development Operations
Start New Development Version	–	Development Operations
Beta Promotion Decision	–	Development Operations
Beta Publishing	Product Owner	Development Operations

*Only sent during Alpha Release Stage.

Table 3.2: Success and Failure Notification Recipients

Job	Google Services Integrations	Implemented By
Main	–	–
Alpha Release Decision	–	–
Alpha Publishing	Google Play Services	Jenkins plugin
Start New Development Version	–	–
Beta Promotion Decision	Google Play Services, Google Analytics	Custom utilities
Beta Publishing	Google Play Services	Jenkins plugin

Table 3.3: Google Services Integration Points

3.2 Jenkins Job Implementation

The following sections describe the implementation of each Jenkins job including relevant and interesting configuration, build tasks, and post-build tasks.

3.2.1 Main Job Implementation

The Main Job is used in both the commit stage and the alpha release stage to build, verify, and package the application. The structure of the main job is shown in Figure 3.3.

The source code repository, in this case Atlassian's Bitbucket service, must be configured to send a "webhook" to the Jenkins server on every commit. This can be secured by the creation of a username/password credential

in Jenkins and including the username, password, and an API token in the URL configured in Bitbucket's webhook settings for the repository. The associated user is given limited privileges in Jenkins. Fine-grain control of the Bitbucket/Jenkins integration is achieved by passing a "notifyCommit" command indicating the repository associated with the commit and setting the Build Trigger of the main job to "Poll SCM" without a schedule.[7]

Gradle is invoked, requesting the `clean`, `check`, and `testDebugUnitTestCoverage` tasks be performed. These tasks will together clear the output and temporary directories within the workspace, build both the debug and release versions, run the tests and generate a coverage report. Test coverage is verified by a shell script Jenkins task that reads the xml file output by the Jacoco test coverage measurement tool.[1] The analysis works by pulling out the total instructions covered and total instructions missed counters, dividing them, and comparing them to a defined threshold. More detailed analysis could be performed using Python or another other language more powerful than shell scripting, if desired. The coverage threshold is configured as a parameter of the Jenkins job, and can be changed for a single run when triggering the job manually.

If the test coverage threshold is met, the Gradle `assembleRelease` task is invoked. This task will package the release build of the application into a signed APK ready for upload to the Google Play Store. The `assembleRelease` task is

[1]Test coverage does not include tests of all types, see Section 5.2.2 for a discussion of this issue.

is invoked from a shell script task in order to allow the signing configuration to be passed in by Jenkins, keeping the signing keys out of source control and therefore more secure [9]. The passwords needed by the signing step are encrypted using the Mask Passwords Jenkins plugin.

A post-build task is responsible for determining whether or not to invoke the Alpha Publishing job. This is set up as a conditional post-build task that checks the source code to determine if the version is marked as a development version.

Finally, the job is configured to send email to the developers and to development operations any time the build fails.

3.2.2 Alpha Release Decision Job

The Alpha Release Decision job is manually invoked, by the Product Owner or a delegate, when release of a given version is requested. The alpha release decision job creates an empty workspace each time it is invoked, to which it downloads the source code from shared source code repository. The alpha release decision job has a single build task runs a Python script in a shell task to remove the "dev" version marking from the appropriate source file and commit the changed file to the local source code repository. If the build steps are successful, the alpha release decision job pushes the source file containing the new version number back to the shared source code repository using the Git Publisher post-build task. In order for the Git Publisher task to work properly, the Jenkins job must be configured to check out and build

a specific branch, and the Git Publisher task must specify the same branch to be pushed to the shared source code repository. The commit to the shared source code repository will result in a webhook call being sent to the Jenkins server, triggering the main Jenkins job. The version number change made by the alpha release decision job will result in the main job calling the alpha release publishing job.

In the event of a failure, the job is configured to send email to the product owner and to development operations.

3.2.3 Alpha Publishing Job

The Alpha Publishing job is triggered when a build of the main job is successful and the version number of the newly-built version is not marked as a development version. The alpha publishing job is configured to use the same workspace as the main job, from which it picks up the signed APK file prepared by the main job. Integration with Google Play Services uses the Google Play Android Publisher Jenkins plugin, which implements an "Upload Android APK to Google Play" post-build task type. To use this plugin to publish to Android Play Services, one must set up a service account in the Google Play Developer Console and in the Google Developers Console. The Google Developers Console will issue a certificate to be used by the Google Play Android Publisher Jenkins plugin, which is added to Jenkins' Credentials page. The Google Play Android Publisher Jenkins plugin will set the build result status to "Failure" if the publishing step encounters an error.

The Alpha Publishing Job will trigger the Start New Development Version job, whether the publishing is successful or not. This is to prevent repeated failed attempts to upload a faulty APK to the Google Play Services.

The Alpha Publishing Job will send an email to the product owner and to development operations in the event of a failure.

3.2.4 Start New Development Version Job

The Start New Development Version job is triggered by every run of the alpha publishing job, whether successful or failing. This job is implemented in the same manner as the alpha release decision job, cleaning a workspace, downloading the latest revision of the application source code, altering the version information and pushing the changed version information back to the shared source code repository. If the start new development version job finds that the version is already marked for development, it will report a failure, as this indicates a potentially dangerous invalid state of the system.

If this job fails for any reason, an email is sent to development operations.

3.2.5 Beta Promotion Decision Job

The Beta Promotion Decision job is responsible for deciding whether there is a version later than the version currently assigned to the beta track that has undergone sufficient testing activity to warrant promotion to the beta channel. Since there is no external event available to trigger this job, it is run

on a schedule. The structure of the beta promotion decision job is shown in Figure 3.4.

The beta promotion decision job uses a small program based on the Google Analytics API and Google Play Services API samples for Java to communicate with these services, and a few Python scripts to assist in processing the data returned. These utilities are stored in a Bitbucket repository, allowing them to be remotely updated if necessary. The job shares a workspace with the Beta Promotion Publishing job, so that the version number to be promoted can be communicated from the beta promotion decision job to the beta promotion publishing job.

This job has one build step which is a bash shell script that invokes the Java and Python utilities, checking for failures at each step. This structure, involving multiple components and three separate languages, is not a matter of design so much as expedience. A more mature implementation of these capabilities might eliminate the use of bash in favor of Python, simplifying Jenkins job configuration and improving handling of failures. Several aspects of the behavior that are hardcoded in the current implementation could be moved to Jenkins job parameters and passed into the scripts and utilities, to give development operations more control without requiring the scripts or utilities to be modified.

The first step of the promotion decision is to retrieve event activity from the Google Analytics service. This connection is secured through a combination of data built into the Java utility and a service certificate like the one

needed by the alpha publishing job. The utility outputs a table of events by version and event type, as described in Section 3.3.2.

The event data is processed by a Python script that filters the table by the screenName value using a regular expression and then totals the totalEvents values for each version. Versions reaching a threshold hard-coded into the Python script are returned to the caller of the script for use in subsequent steps of the decision process. This is a very simple implementation; more complex analysis could be easily implemented, for instance to look for minimum values across a number of different screenNames, eventCategories, or eventActions for each version.

The next step of the beta decision process is to determine what versions are assigned to which tracks. The Java utility calls the Google Play Services API to retrieve this data, using the same authentication and authorization approach as the previous call to the Google Analytics API. An effective beta version is determined by checking for an APK assigned to the beta track, and if there is none, an APK assigned to the production track. If no APK is assigned to either track, the effective beta version code is set to the empty string, ensuring that no version will be passed over for promotion due to its version number.

Finally, a promotable version number is identified if the list of versions passing the activity threshold contains any version number higher than the effective beta version number. If a promotable version number is found, it is written to a text file in the workspace, to allow it to be injected into the build's

execution environment as an environment variable named `$promotableVersion` for use in post-build steps and in the beta promotion publishing job. If no promotable version is found, this environment variable is set to the value "none".

The promotion decision is finally executed by a post-build job that checks the value of the `$promotableVersion` environment variable. If this variable's value consists only of digits, the beta promotion publishing job is invoked.

The beta promotion decision job is configured to send email to development operations in the event of a failure. Successful processing of the data that results in a decision not to promote any version to beta is not considered a failure.

3.2.6 Beta Promotion Publishing Job

The Beta Promotion Publishing job uses the "Move Android APKs to a different release track" task implemented by the Google Play Android Publisher Jenkins plugin to assign a version to the beta track. This job shares a workspace with the beta promotion decision job. The first build step of the beta promotion publishing job is to retrieve the `$promotableVersion` environment variable. The next build step is the Move Android APKs step, which takes the application's application ID and version code and uses the same credentials used by the Upload Android APK task in the alpha release publishing job.

The Beta Promotion Publishing job notifies development operations of build failures via email. Email is sent to the product owner in the event of a successful promotion of a new version to the beta track.

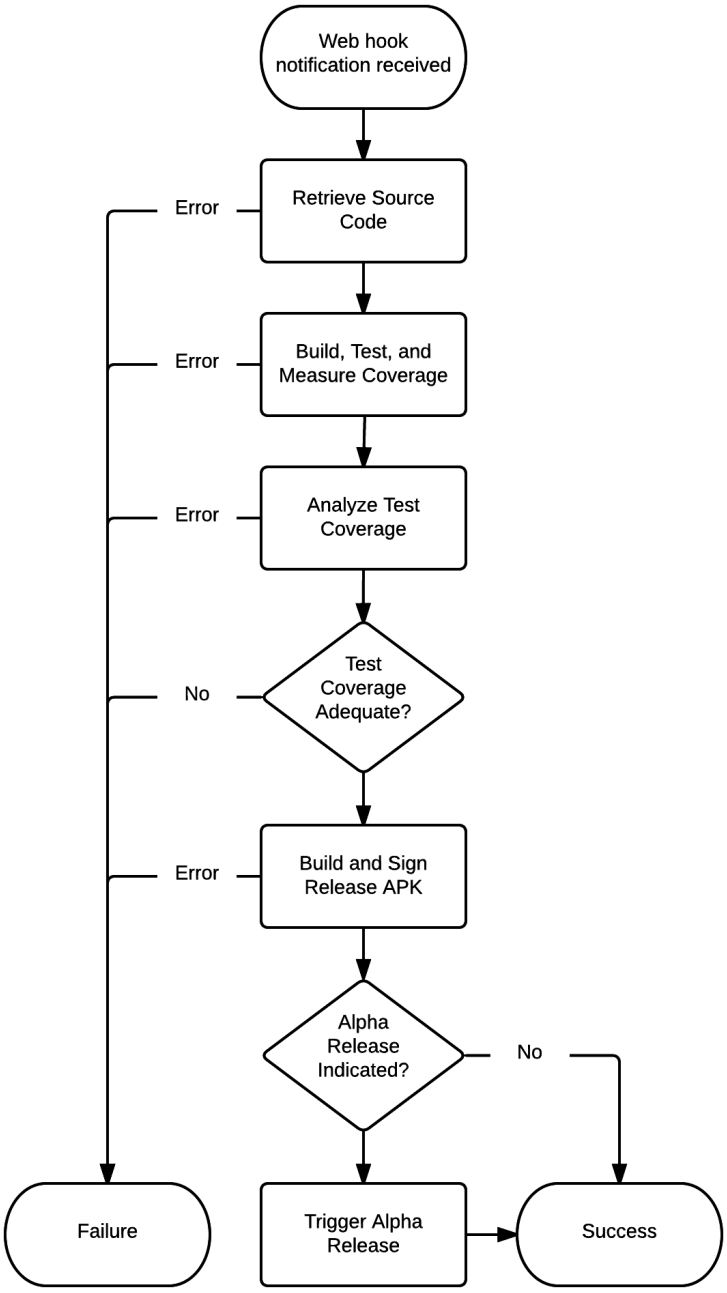

Figure 3.3: Main Application Build Job Structure

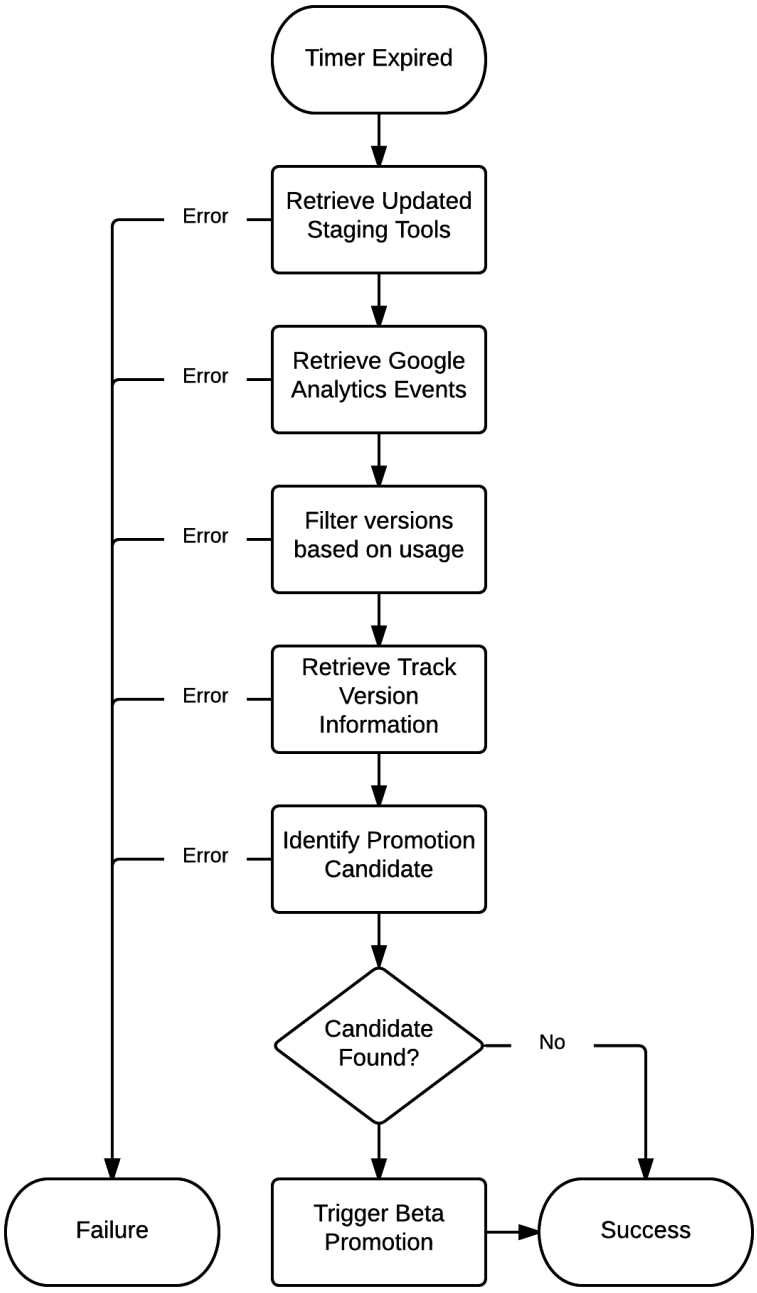

Figure 3.4: Beta Promotion Decision Job Structure

3.3 Custom Google Service Integration

Custom Google service integrations were implemented in the Android application as part of the Manual Testing pipeline stage and in utilities used by the Beta Promotion Decision Jenkins job. These custom integrations together cooperate to support the automatic promotion of an APK that has seen sufficient testing activity during the Manual Testing phase to the Play Store's beta track.

3.3.1 Google Analytics for Android

The Android application uses Google Analytics integration to register user activity with Google Analytics. Integration based on Google's documentation and sample code is straightforward; if further information is needed, *Android Cookbook* contains detailed steps for implementing this integration [4]. Android project setup steps include:

- Ensuring that the Google Play Services SDK package is installed.
- Adding a `com.google.gms:google-services` dependency to the project-level build.gradle script.
- Adding the `com.google.gms:google-services` plugin to the application-level build.gradle script.
- Adding necessary configuration files.

Interactions with the Google Analytics service use a tracker object which is instantiated with a call to `GoogleAnalytics.getTracker()`. Google's documentation recommends caching this object in a custom Application object. Once this is accomplished, any location in the code that can make a call to `getApplication()` can send activity to Google Analytics.

Each event to be sent to Google Analytics is modeled as a "Hit". Each hit has a variety of attributes that can be set, including "screenName", "category", and "action". These can be set to whatever values best serve the purposes for which the data is being collected. I used an Android Activity's title for the screenName, "Method" for the category to indicate that the event corresponds to a method call within the Activity, and the name of the method being instrumented for the action.

Calls to Google Analytics are disabled by default in debug builds. One can use the Android Debug Bridge (adb) to enable debugging of Google Analytics calls by sending the following command to the connected device:

```
adb shell setprop log.tag.GAv4 DEBUG
```

Once the hit has been registered, it is queued for delivery on a separate thread. Depending on the environment, and assuming constant network availability, it may be between two and 30 minutes before it is sent from the device to Google Analytics. Hits that cannot be sent to Google Analytics by 4 AM of the following calendar day will be discarded [1].

The distinction between development usage and alpha usage for the same version is critical to ensuring validity of the decisions made on the basis

of this data. This was addressed by directing the Google Analytics integration code in the Android application to check the version information available at run time and report the "versionName" to Google Analytics if the version information indicates a development version, and the "versionCode" attribute otherwise.

3.3.2 Google Analytics Utility

To use the Google Analytics data sent by the Android application from Jenkins, I implemented a custom Java utility class that is invokable from the shell. This class makes use of the Google Analytics API for Java. The utility uses a service account and key file for authentication to and authorization by the Google Analytics service in the cloud. This is a similar scheme to that implemented by the Google Play Android Publisher Jenkins plugin used in the Alpha Release Publishing and Beta Promotion Publishing Jenkins jobs.

Aside from properly initializing the Google Analytics API for secure access to the application's recorded events, retrieving data is a matter of specifying the data desired and parsing the results returned. The data to be retrieved is specified in terms of a date range, one or more "dimensions", and one or more "metrics". The results returned by a query will have at most one row per unique combination of specified dimensions. Each row will have a column for each dimension and a column for each metric. Metrics represent aggregated values, and are reported for each combination of the specified dimensions that is found in the application's Google Analytics data within the

```
return analytics.data().ga()
  .get("ga:" + profileId, "7daysAgo", "today", "ga:totalEvents")
  .setDimensions("ga:appVersion,ga:screenName,ga:eventCategory,ga:
     eventAction")
  .execute();
```

Figure 3.5: Retrieving Event Data from Google Analytics

```
SELECT appVersion, screenName, eventCategory, eventAction,
       COUNT(events)
FROM analyticsData
WHERE profileId = [myProfileId]
  AND eventDate > [sevenDaysAgo]
  AND eventDate < [today]
GROUP BY appVersion, screenName, eventCategory, eventAction
```

Figure 3.6: SQL Translation of Google Analytics Retrieval Call

date range. By analogy to SQL, the dimensions are like SQL `GROUP BY` fields and the metrics like SQL aggregate functions. The call I used for retrieving event data from Google Analytics is shown in Figure 3.5. Figure 3.6 shows what this call might look like if translated into SQL.

For my current purposes, I wanted to separate the Google Analytics integration code from the code that makes decisions based on the data. To support this, the Google Analytics utility writes the data out to a tabular file for processing by a Python script, as described above in Section 3.2.5.

```
TracksListResponse response = edits.tracks()
    .list(ApplicationConfig.PACKAGE_NAME,appEdit.getId())
    .execute();
```

Figure 3.7: Google Play Services API Track Listing

3.3.3 Google Play Services Utility

The final piece of data needed to decide whether to promote a version to the beta track is the version currently in beta. To determine this, I implemented a custom Java utility class, like the one implemented for Google Analytics, but using the Google Play Services API. This class uses the same security implementation as the Google Analytics utility; the differences are in the calls to retrieve the data and in the format of the data retrieved.

While the Google Analytics API exposes a small number of methods and the data to be retrieved is specified as arguments to those methods, the Google Play Services API for Java exposes a relatively larger number of methods, each of which has few if any arguments. The data needed for the implemented continuous delivery pipeline is nothing more than a list of the APKs assigned to each release track. This is retrieved by making the call shown in Figure 3.7

The tracks list response contains a series of `Track` objects, each with a name and one or more version identifiers for the APKs assigned to that track. APKs are distinguished by their versionCode value, which is an integer that must increase with each newly uploaded APK. As with the Google Analytics data, I wanted to separate retrieval from processing and so elected to write

the track and version mapping out to a tabular file for processing by a Python script.

Chapter 4

Results and Findings

4.1 Results

Starting with little more experience with continuous integration than "Hello World" and only a little more experience than that with Android, I was able to implement a continuous delivery pipeline for an Android application covering build, test, analysis, and deployment tasks for an Android application. This pipeline would allow the members of a team responsible for delivering such an application to concentrate on their roles' primary responsibilities by removing the distraction and drudgery of manually performing procedures not directly related to their job duties.

4.1.1 Resulting Development Operations Dashboard

The suite of continuous integration jobs that I implemented not only performs the necessary tasks, but does so in a way that would allow those acting in the development operations role to understand the state of the delivery pipeline by understanding the state of the jobs involved, as shown in Figure 4.1.

The default Jenkins dashboard configuration shows the final status of

Figure 4.1: Jenkins Dashboard for Implemented Continuous Delivery Pipeline

the most recent build (blue circles at left, indicating success), recent trend using a "weather" indicator (sunny = consistently good, rainy = failing, and so on), as well as dates of the most recent successful and failing builds and duration of the most recent build.

By reading the dashboard in Figure 4.1, development operations can determine the following:

- Release of a new version of the application to the alpha track was requested 2 days and 11 hours ago, and successfully completed shortly thereafter.
- Manual testers completed necessary testing of the alpha versions 7 hours or so after the release to the alpha track.
- The release was promoted to beta 2 days and 3 hours ago.
- No changes have been committed to the application's source code repository by developers in over 2 days.

S	W	Name	Last Success ↑	Last Failure	Last Duration
		LanIrAndroid	8 min 16 sec - #138	5 days 10 hr - #128	1 min 17 sec
		LanIrStartDevelopmentVersion	8 min 56 sec - #15	4 days 14 hr - #11	6.9 sec
		LanIrAlphaReleaseDecision	11 min - #25	8 days 11 hr - #8	11 sec
		LanIrBetaPromotionDecision	58 min - #234	N/A	6.1 sec
		LanIrBetaPromotion	2 days 3 hr - #202	8 days 15 hr - #196	9.6 sec
		LanIrAlphaRelease	2 days 11 hr - #16	9 min 6 sec - #17	12 sec

Figure 4.2: Jenkins Dashboard Showing Failed Alpha Release

Figure 4.2 shows a different state of affairs. This dashboard indicates the following changes from the conditions in Figure 4.1:

- Release of a new version of the application to the alpha track failed about 10 minutes ago.
- The release was requested in the normal manner shortly before the release failed.
- The problem was encountered in the course of attempting to upload the new APK file, not in building, testing, or packaging the application.
- Accidental attempts to release to alpha are unlikely, as the application's version number has been marked for development in the source control repository.

In addition to being transparent to its maintainers in development operations, the implemented continuous delivery pipeline is unobtrusive and re-

sponsive from the perspective of the user roles considered in its design. Developers need only commit their changes to the shared source code repository to rehearse packaging of the product, including verification of test code coverage. Notifications of broken builds are delivered via email within minutes of the breaking commit. The product owner is able to request release of the current version to alpha and be kept informed of the staging progress through the alpha release and manual testing stages to beta release. Manual testers need only complete the steps of the test to mark the tests successful; if tests fail, they have time to "stop the presses" before the failing version will be promoted to beta.

4.1.2 Effort Involved

Implementing the continuous delivery pipeline, including the implementation of test coverage verification as well as Google Services integration, required a surprisingly small amount of custom code. I implemented custom Java code for Android application integration with Google Analytics and Jenkins integration with Google Analytics and Google Play Services. I wrote Python scripts to perform data manipulation and analysis. I used shell scripts to specify workflow within Jenkins jobs and to calculate code coverage as a percent of lines covered. Table 4.1 shows the line counts for each of these components.

The greatest part of the time spent implementing the continuous delivery pipeline was in working out difficulties in integration of code coverage

tools with the rapidly-changing state of the standard Gradle build plugin for Android. In the end, this effort was only partially successful; tests relying on dependency injection frameworks like PowerMock are not currently included in the coverage metrics available to the Jenkins job. See Section 5.2.2 for more on this issue.

The importance of a transparent dashboard for development operations use, and the need to model the continuous delivery pipeline as shown in the figures and tables in Chapter 3 to produce such a dashboard, only became apparent after sustained use of the pipeline to perform multiple releases. Section 4.2.3, below, expands on this subject.

I would estimate that implementing a similar pipeline for an Android application would take between one and two weeks, assuming use of the same versions of the tools involved. Some of this time would be in manual configuration of the Android Service accounts required for publishing and analytics components, and in manually creating the initial product listing in the Play Store. If the scenario described at the beginning of Chapter 3 did not apply as-is, the time to implement the pipeline would be extended to allow time for analysis of the software development organization's business processes, establishment of an appropriate set of requirements for the continuous delivery pipeline, and modeling of the new pipeline's desired behavior.

Language Component Subcomponent	Lines
Java (line counts based on IntelliJ's internal code coverage tool)	
Google Analytics for Android	
Per application	6
Per activity	2
Per Google Analytics event	2
Google Services Utility for Jenkins	
Retrieve Google Analytics Events	56
Retrieve Google Play Services Tracks/APKs	29
Google Services Shared code	2
Python (not including imports, comments, or blank lines)	
Google Services data analysis	40
Version number updates	
Update version for release	11
Update version for development	31
Shell scripts (not including comments or blank lines)	
Beta Promotion Decision	34
Calculate test coverage	7
Version number updates	
Update version for release	6
Update version for development	6
Total	232

Table 4.1: Lines of Code by Component

4.2 Findings

4.2.1 Integration With Google Services

Integration with Google Analytics was easier than I expected it to be, both in the Android application and in the utility used by the beta promotion decision Jenkins job. The same was true of the Google Play Services integration utility. The main challenge in using these services as intended was in mapping from the version numbers as reported by Google Analytics to the version numbers as reported by Google Play Services. This was resolved by altering the Android application's Google Analytics integration from the default behavior to report the versionCode rather than versionName to Google Analytics, as described in Section 3.3.1.

The first attempts at building the Android application with Google Analytics support using Jenkins failed with a dependency error relating to `com.google.android.gms:play-services-measurement:8.1.0`. I traced this to the fact that I had installed the Android SDK in a separate location for Jenkins. As a result of this decision, I was required to install the Google Play Services SDK package to this location as well as to the location used by Android Studio.

Full unit testing of the Android application's Google Analytics integration capabilities would have required building a spy object for the Google Analytics `Tracker` class and verifying that the correct calls were made to it. While I did not want to go to this trouble for this project, I did want to ensure that the Google Analytics configuration was being loaded properly. The `Tracker`

class used within the Android application exposes a generic `get(string key)` method for retrieving these properties as though from a key-value map. The class does not expose a collection of valid keys, and the key names are not easy to find in Google's documentation. I was able to determine the correct keys to use by using the Java Bytecode Decompiler plugin in Android Studio to view the decompiled `Tracker` class.

4.2.2 Latencies in Google Services

Google Analytics and Play Services are global services with hundreds of millions of active users. Distribution of data across these services' infrastructure is not instantaneous. This can slow responsiveness of those parts of the system that rely on these services. In the case of the implemented continuous delivery pipeline, these delays are seen at the start and completion of the manual testing stage. These delays would be very damaging to a continuous delivery process that relied on the affected systems as part of a test-driven development cycle; waiting an hour or more to get an update onto a device and 15 minutes to get the results of testing would be unacceptable. As relates to verifying that the product can be successfully published through the Android Play Store, the implemented pipeline performs adequately for circumstances other than critical emergency releases.

4.2.3 Importance of Single-transaction Jenkins Jobs

During implementation of the continuous delivery pipeline, I often found myself, acting in the Development Operations role, confused as to what the current state of the system was. I had implemented the Main and Alpha Release Publishing jobs as a single job, and it was not clear from reading the Jenkins dashboard how recently a new release had been deployed to the Google Play Store. The Beta release stage was implemented as a single job, which controlled whether to attempt promotion by failing if no promotable release was found. This made it impossible to distinguish from the dashboard whether promotion had been attempted and failed or simply not been attempted. As I worked to understand the Development Operations role's needs for system status information, it occurred to me that I had not considered which notifications should be sent to which roles, since I was playing all of the roles. These problems and others like them made it apparent that transparency of the pipeline to development operations and others would require careful modeling of the processes within each stage.

This modeling, reflected in the tables and figures included in Chapter 3, led toward a one-to-one mapping between continuous integration jobs and affected global system state. That is to say, each job is responsible for making changes to at most one shared data resource, as shown in Figure 4.2. Two of the jobs do not change a shared data resource other than the local workspace. These jobs use this mechanism to pass data to downstream jobs sharing the workspace.

Jenkins Job	Shared Data Resource Affected
Main	(local workspace)
Alpha Release Decision	Source code repository
Alpha Release Publishing	Android Play Store
Start New Development Version	Source code repository
Beta Promotion Decision	(local workspace)
Beta Promotion Publishing	Android Play Store

Table 4.2: Shared Data Resources Affected by Continuous Integration Jobs

This one-to-one mapping causes each job to more or less closely correspond to a single transaction that can be said to have succeeded or failed. Control flows can be created by chaining of related jobs using conditions without compromising the ability of development operations to quickly comprehend global system state. This technique also allows reuse of common functionality at the level of the continuous integration job.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

I was able to demonstrate that continuous delivery of an Android product is a practical goal. Implementation of such a pipeline should require only a few days or, at most, a few weeks of effort. A lot of this effort might be in understanding the needs of the organization relative to pipeline control points and status visibility. The organization would likely benefit from this understanding whether the continuous delivery pipeline resembled the pipeline implemented for this project or not.

I was also able to demonstrate that Google Analytics event tracking can be used to automatically monitor and react to manual tester activity. This automation and the use of it to make staging decisions supports a "look before you leap" approach to deployment into outside testing groups, acting as a gatekeeper to prevent faulty or broken deployments and upgrades from impacting beta testers.

In practical terms, I would recommend that an organization using a pipeline like the implemented pipeline expect to perform alpha releases no more often than once per day. The release could be requested at the end of

the day to allow time for Google Play Services to distribute the update to the manual tester's client devices. If the manual tests were short enough, the beta promotion could be expected to occur early in the next day, allowing the beta testers to access the updated version within 24 hours of the request to release it.

Application of automated staging based on Google Analytics data to the final promotion of an APK to general availability or GA is a natural progression from the implemented pipeline. Fully automating this decision might require much more detailed analysis of events, including verification of correct behavior rather than the simple "hit counting" approach used in this project. If full automation is not the goal, an organization could implement a Jenkins job to notify the Product Owner of versions that have seen enough successful usage while in the beta track to be candidates for promotion to GA, and another Jenkins job to perform promotion to GA at the press of a single button.

This approach to staging decisions is equally applicable to systems using user activity tracking capabilities other than Google Analytics. Systems that include centralized databases could store the necessary event data in separate tables. Many systems, such as workflow systems or other online transaction processing (OLTP) systems, could infer activity from events already being tracked in the OLTP databases, if appropriate version information could be related to user activity. The approach could also be applied to other types of products and other deployment mechanisms, for instance, web servers using

the Node.js Package Manager.

Integration of user activity data into a continuous integration pipeline opens a wide vista of possibilities regarding automation of staging decisions. This work has implications beyond Android applications and Google Analytics to any application for which instrumentation data can be aggregated and analyzed.

5.2 Future Work

The most obvious and to me interesting future work is that described above in Section 5.1, to expand the breadth of instrumentation and the depth of the analysis to make better decisions based on manual tester activity. The following sections describe a few other possible directions of further work in this area.

5.2.1 Test Coverage Threshold Ratchet

The implementation of a minimum test coverage threshold is limited to a default threshold and a per-build manual override. In a development organization starting without adequate automated test coverage and without a culture of test-driven development, we might expect to see coverage rise as tests are written and fall as new code is written without accompanying tests. This state of affairs could be improved by automatically raising the minimum test coverage threshold whenever actual test coverage exceeded the minimum threshold. There are a number of approaches to this, among them (a) altering

the Jenkins job's configuration automatically, and (b) storing the test coverage threshold in source control or elsewhere outside of Jenkins.

5.2.2 PowerMock Coverage from Gradle

An unmet goal of this project was the measurement of test coverage for all types of automated tests. One thing I was not able to get working was inclusion of coverage information for tests using PowerMock. The reason for this is that PowerMock accomplishes the necessary dependency injection by re-writing portions of the class path in ways that produce data that is not properly handled by Jacoco. Addressing this deficiency might involve research into the Gradle plugins and tasks that define the build process to allow PowerMock and Jacoco to work together, or the creation of a data manipulation capability that can resolve the incompatibilities between these two tools.

5.2.3 Automated Acceptance Testing

The primary goal in implementing automated tests for this project was ensuring technical compatibility of the tests with the various development and execution environments. This included "white box" test methodologies like JUnit and Mockito that would be appropriate for pre-commit "unit testing" [1] as well as "gray box" test methodologies like Robolectric and Android Instru-

[1] "Unit" testing has a specific meaning, denoting the testing of precisely one unit of code and its expected functionality. Oftentimes, the testing performed by developers prior to committing to the shared source code repository is not strictly speaking "unit" testing, but is instead partial integration testing. It is common for developers to refer, incorrectly, to these tests as "unit tests".

mentation Tests that would be more appropriate for automated acceptance testing. Implementation of an automated acceptance testing stage would primarily be a matter of test case and test suite management and the creation of continuous integration jobs to run the automated acceptance tests and respond to the results.

5.2.4 Automated Device Compatibility Testing

The authors of *Continuous Delivery* discuss the performance testing stage primarily in terms of capacity testing. Capacity testing is not a particularly relevant topic for Android applications in specific and user interfaces in general. For an Android application, perhaps the most important performance concern is device compatibility; there are far too many different types of devices for most development organizations to be able to purchase even a representative sampling of devices for compatibility testing. An Android Device Compatibility Testing stage could be implemented using a specialized emulation server or farm of such servers. There are naturally questions about the fidelity of emulators. Due to this, ensuring test validity and accuracy would be critical if this strategy were pursued. Google is offering a service called Google Cloud Test Lab that automatically performs testing on a variety of physical devices. If I were developing an Android application professionally, I would treat the Google Cloud Test Lab as my primary strategy for accomplishing device compatibility testing.

Bibliography

[1] Dispatching - Android SDK v4. https://developers.google.com/analytics/devguides/collection/android/v4/dispatch, 2015. [Online; access 17-Nov-2015].

[2] Our history in depth. http://www.google.com/about/company/history/, 2015. [Online; access 14-Nov-2015].

[3] Brian Clifton. *Advanced Web Metrics with Google Analytics*. Sybex, third edition, 2012.

[4] Ian F. Darwin. *Android Cookbook*. O'Reilly Media, 2012.

[5] Jez Humble and David Farley. *Continuous Delivery*. Addison-Wesley, 2011.

[6] Jeff Carollo James A. Whittaker, Jason Arbon. *How Google Tests Software*. Addison-Wesely Professional, 2012.

[7] Kohsuke Kawaguchi. Polling must die: triggering Jenkins builds from a git hook. http://kohsuke.org/2011/12/01/polling-must-die-triggering-jenkins-builds-from-a-git-hook/, 2011. [Online; access 14-Nov-2015].

[8] Daniel Knott. *Hands-On Mobile App Testing*. Addison-Wesley Professional, 2015.

[9] Matthew Mehalso. Pro Tips for Continuous Integration with Android and Jenkins. `http://bluefletch.com/blog/pro-tips-for-continuous-integration-with-android-and-jenkins/`, 2015.03.23. [Online; access 14-Nov-2015].

[10] Ken Schwaber. *Agile Software Development with Scrum*. Microsoft Press, 2004.

[11] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2002.

[12] John Ferguson Smart. *Jenkins: The Definitive Guide*. O'Reilly Media, 2011.

[13] Jeff Sutherland. *Scrum*. Crown Business, 2014.